Københavns Universitets Matematiske Institut
Afdelingen for Informationsbehandling
Universitetsparken 5.

I M P III

by

B. Svejgaard and P. Lindblad.

K.U.A.I.B. 10.

I M P

(Interpretative Macro Processor) is a program for the generation of symbol strings. It may be used for writing programs in any programming language for any machine, as well as for writing file cards, business letters ect.

The substantial part of the IMP-language is the macro. The macro is defined by the macro-definition (see syntax). The brackets [and] embracing the macro definition will have effects as the begin, end in ALGOL, thus shielding macro-names declared inside a macro-definition from the outside.

The macro-definition itself may be looked upon as a parallel to the procedure-declaration in ALGOL. It consists of two parts of which the first part, the name, may be any sequence of symbols aside from the delimiters particular to the IMP-language. The second part, the macro-string, is a symbol-string which itself may be written in IMP-language. Besides it may contain formal parameters.

The active part of the IMP-language is the macro-call indicated by the brackets (and). The symbol-string appearing between (and * or between (and) is the name of the macro-call. The following strings between * and * or between * and) are the actual parameters. These may again contain formal parameters, calls and definitions.
During the processing the macro-call will be replaced by the macro-string appearing in the macro-definition, having the corresponding name, and valid at the appropriate level. The formal parameters of the macro-string will be replaced by the actual parameters in the macro-call. The integer representing each formal parameter will indicate, which of the actual parameters to use, the actual parameters being numbered from the left to the right.

Macro-definitions and macro-calls may appear in any order. However a macro must be defined not later than by exit of the macro-definition of highest level inside which it is called.

Dummies will be left out in the resulting string, but the strings of symbols occurring between ₹ and the following ₵ will be outputted on reading time.

IMP-syntax

<delimiter>::= [|] | (|) | * | = | ¢ | ₹ | £
<dummy>::= CR | SP |<comment>
<symbol>::= any other character alone or with underline or vertical bar |<dummy>
<comment>::= ¢<basic string>£|₹<basic string>£
<basic string>::= <symbol>|<basic string><symbol>
<macro call>::= (<macro name><parameter list>)|(<macro name>)
<macro name>::= <basic string>
<parameter list>::= <actual parameter>|
<parameter list><actual parameter>
<actual parameter>::= *<compos.string>|*
<compos.string>::= <macro call>|<basic string>|<macro definition>|
<formal parameter>|<compos.string><compos.string>
<macro definition>::= [<macro name>*<compos.string>]_G<macro name>*]
<formal parameter>::= =<positive integer>=
<positive integer>::= <proper digit>|<positive integer><digit>
<proper digit>::= 1|2|3|4|5|6|7|8|9
<digit>::= 0|<proper digit>

In handwriting the somewhat clumsy delimiters may be written as follows:

[and] as ⌈ and ⌉,

(and) as ⟨ and ⟩,

* as × and

= as /.

The delimiters may be redefined currently. In this way it is possible to treat an IMP-string by an IMP-program, and to give an IMP-program a nicer look. (See the section: Operating the Processor.)

The use of IMP

The idea of IMP is not to supply the user with a set of standard macros, but to give him the possibility of defining his own macros.

In this way he will not be forced to stick to conventions made up by other people, but he may define a language suitable for his particular problem.

A few examples may serve to illustrate the use of macros.

In a certain program part (which may be defined as a macro) it is desired to calculate the values of different polynomials for certain arguments. Then, during the writing we assume the existence of a macro ,poly, with two parameters, the first giving the (address of the) highest order coefficient and the second the (address of the) argument. The constant term is supposed to be a-marked.

The generation of the necessary commands then may be executed by calls like

```
(poly * c1 * a2)
```

or

```
(poly * (coeff.) * (arg))
```

In the latter case the macros ,coeff, and ,arg, must be defined elsewhere e.g. by

```
[coeff * c1] [arg * a2]
```

If, by looking through the program-part under consideration, we find that the macro ,poly, has been called only very few times, we may decide to define it as an open subroutine by

```
[poly *
 panr1t=1=
 arX1
 qqVXLA
 mk =2= , hv r-2]
```

(NB: The cr following * is a part of the macro-string defined.)

However, if the macro is called very often, space may be saved by defining the macro by

```
[poly *
 panD =1=
 hs n e0
 qq =2= ],
```

If at the same time we have a set of instructions

```
e0: ar(s-1)X1
    hrs2XLA
    mk(s1),hvr-2.
```

This set of instruction may again be generated by a call (polrut * e0) referring to a library macro

```
[polrut *
 =1= : ar(s-1)X1
       hrs2XLA
       MK(s1),hvr-2].
```

If the following macros have been defined

```
[val * c3]
[calc * c4]
[result * qq c8]
[comp. *
 arn (val)
 hs  (calc)
 (result)],
```

then each call (comp.) will generate the string

```
arn c3
hs e4
qq c8.
```

Had ,val, been defined by [val * c =1=] and ,comp, by [comp *

```
arn (val * =1=)
hs (calc)
(result)]
```

then the same string would have been obtained by the call

```
(comp * 3)
```

The definition

```
[exc *
 =1= rn a1
 hs a2]
```

will permit calls as

```
(exc * s)
```

and

```
(exc * a)
```

generating respectively

```
srn a1
hs a2,
```

and

```
arn a1
hs a2.
```

The same result may be obtained by the definition

```
[exc*rn a1
 hs a2]
```

and the calls

```
s(exc)
```

and

```
a(exc).
```

In Algol IMP may sometimes be used to avoid the use of procedures if execution time is essential.

The macro

```
[cm *
=3=[0]:= =1=[0]×=2=[0] - =1=[1]×=2=[1];
=3=[1]:= =1=[0]×=2=[1] + =1=[1]×=2=[0];]
```

may be used for the multiplication of two complex numbers each represented as arrays as X[0:1]. Thus the call

```
(cm * alpha * b * r)
```

will generate the piece of algol string necessary to form the complex number r = alpha.b

If the following macro exists

```
[invit*
 It would be the pleasure of =1= to see =2=
 at an informal party on =3= at =4=.
],
```

then the definition

```
[I * (invit * Regnecentralen *
  =1= * dec. 12. 1964 * 7.30 p.m)]
```

and the calls

```
(I * name1)
(I * name2)
(I * name3)
    .
    .
    .
```

will generate a set of invitations to each of the persons name 1, name 2, ...

The following example will show how symbolic SLIP-addresses may be replaced by more suggestive names.

```
[]<>/-crE
[M/b a12
<start>:    vy1, sy64
            lyn<char>, nc64[char/a8]
<minus>:    hvr-1 [minus/a7]
<char>:     lyn<char> D
            ca16, grn<count>[count/a1]
            nc, hh<char>
<read>:     vy1.4+3.9, lyn<char>[read/a2]
            hv<par> NT[par/a3]
            mbr1, ca15
            sy15, hh<read>
            mt<minus>, arr-1
            gar1
            sy, hh<read>
<par>:      vy1.5+1.9, lyn<char>
            nc16, hv<write>[write/a5]
            arn<one>, ac<count>[one/a4]
            hv<read>
<write>:    nc<p>, hv<par>[p/39]
            pm<count>, pa<zero>[zero/a11]
            par1t<const>[const/a9]
<div>:      dlnt1 IRD[div/a6]
            pa<zero> t1G NZ
            tk30, ga<out> [out/a12]
<zero>:     sy LZV
<out>:      sy
            hv<div> NRD
<const>:    sy64, hv<start>[start/a10]
            100000
             10000
              1000
               100
                10
<one>:           1b
<count>:    qq
            e10]
```

The following macros will sometimes be useful in machine programming.

```
[REP*      it=3=, par=2=
=1=:        =4=
=2=:       btt-1
           hvr=1=]
```

Here the parameters =1= and =2= are suitable chosen symbolic addresses and macros representing symbolic addresses. The parameter =3= indicates how many times the piece of program represented by the parameter =4= shall be repeated. It may of course be an indirect address. Thus in a certain program the lines

```
a3:        (REP*a1*a2*3*
           qq t1
           paa4 t1
           (REP*a5*a6*12*
           (mov*(ref)*(pc)*a)
a4:        arn(a), c1 10
           arn D t1
           c1 20
           arn(a3) D
           c1 -30, gr(a)))
```

have been replaced by the commands

```
a3:        it3, para2
a1:        qqt1
           paa4 t1
           it12, para6
a5:        arn(1d), gaa
           arn(a), ga(1d)
           arn(3d), ga(a)
           arn a, ga(3d)
           arn a, c1 10
a4:        arn D t1
           c1 20
           arn(a3) D
           c1 -30, gr(a)
a6:        bt t-1
           hvra5
a2:        bt t-1
           hvra1
```

The macro

```
[outsym*    par2t=1=
            sy=2=, it-1
            bt, hvr-1]
```

will generate the commands necessary to output the number =1= of the symbol =2=.

Thus (outsym*70*0) will output 70 spaces.

The FORTRAN IF may be simulated by the macro

```
[IF*  =4=
      hvr=1=LT
      hvr=2=LZ
      hvr=3=]
```

where =4= represents a piece of code leaving a result in the R-register, and =1=, =2=, =3= are addresses to be jumped to, if this result is respectively negative, zero or positive.

To see how subroutines may be handled let us take the SLIP-routine for outputting texts. It may be made a macro thus:

```
[SLIPTEXT*_
(SLIPLAB):  it(s1), par2
            it0
             .
             .
             .
            etc.]

[text* hs(SLIPLAB)
       qq=1=]
```

If in the program we have written: (SLIPTEXT)[SLIPLAB*e8], the SLIP-routine will be placed here and will get the address e8 for its first command. Now any writing of texts may be effected by calls like:

(text*a0) or (text*(explanation)).

This mecanism may be even more efficient by means of the macro

```
[outtext*  hs(SLIPLAD)
           qq(textadr)
           (temp)=i, i=(textadr)
           t=1=;
           (textadr)=i, i=(temp)],
```

where (temp) and (textadr) have been defined as symbolic addresses. In this case every call like

```
(outtext*Place tape in reader)
```

will cause the textstrings occurring as parameters to be stored consecutively and will generate the jump command and the following qq command necessary for the writing of the parameter strings.

The proper choice of delimiters may serve to replace clumsy constructions as

```
(elem*(pc)*a)
(post*a*c)
(sub*b*d)
(sub*c*(val))
```

by

```
 elem, pc;,a; post,a,c; sub,b,d; sub,c, val;;
```

where space have been used for (, ; for), and , for *.

It should perhaps be pointed out, that a formal parameter for one macro may be an actual parameter for another. Thus the two macrodefinitions

```
[SHD*  hs=1=
       qq=2=.13+=3=.31]

[reset* (SHD* e7*=1=*=2=)]
```

are equivalent to the single definition

```
[reset* hse7
        qq=1=.13+=2=.31].
```

Due to the block structure any program-part written in IMP may be preceeded by [and a name followed by * and terminated by]. It may then be called to define following macros or a macro embracing it.

Operating the Processor

The IMP processor is published as a HJÆLP-routine, himp, and as a parent-tape for producing HJÆLP independent versions of IMP. himp is loaded as a normal HJÆLP-routine and occupies 18 drumtracks. It may be called with no parameter or with one parameter. The parameter should be qq<a>, where <a> is 1 + the last track number which may be used by the processor. A call with no parameter is equivalent to a call with parameter qq320, (qq960 on the version to be used on GIER with 3 drums).

The parent-tape is loaded by typing 1 (SLIP). After a few lines have been read, the reader will stop, and the SLIP-addresses c and c1 can be redefined from the typewriter. c is the track number of the first of 17 tracks for storing the processor and c1 has the same meaning as the himp parameter. After redefinition the loading is continued by typing 1. When a kompud version of the processor is made, it should consist of cells 10-21 of the core store and drum-tracks c to c+16. The program starts in cell 10.

When the message IMP has been typed, the processor may be started by typing a space. Every symbol appearing on the input tape before the first [is ignored. Input is continued until the] corresponding to the first [is read or until an end-code (flexo-writer code 12) is read. In the latter case the message wait is typed and input is continued when a space is typed.

When the processor is started or restarted after the wait message by a space the set of delimiters is unchanged (from the beginning they are those mentioned in the syntax). By typing d (delimiter definition) or a t (type delimiter definition) the delimiters can be redefined. In this case the first 11 symbols (an underlined symbol or a symbol with a vertical bar is considered as one symbol) appearing on the input tape (d) or typed on the typewriter (t) are used as substitutes for the original delimiters in the following order:

[ ] ( ) * = ¢ ‰ ¤ <wait symbol> <take new delimiters>

The appearance of the delimiter <take new delimiters> on the input tape will cause the 11 symbols following it to be used as delimiters in what follows.

If the same symbol appears twice in the set of delimiters it will be effective only in the first meaning. The dummies, underlined space and carriage return may also be redefined to be delimiters. (In the standard set of delimiters end-code is used as <wait symbol> as well as for <take new delimiters> that is the latter is undefined).

During a normal run KA and KB must be set to 0. KB=1 will cause the processor to stop between its 3 passes. (To restart the processor press the normal start button). KA=1 will cause the processor to print the internal output from the passes.

When a run is completed, and the value of the outermost macro has been printed on tape, the message IMP is typed, indicating that the processor is ready for a new run.

Comments beginning with an ℝ (report) are typed on the typewriter during input.

If a syntactical error is encountered on the input tape a message consisting of 3 numbers (in red) is typed. The first number is the linenumber (counting the line containing the first [as line 0). The following two numbers are row and column index, respectively, of the syntactical matrix. The rows caracterises the (forbidden) delimiter and the columns caracterises the context in which the error is found. [meaning the macroname-part of a macro definition, (* meaning the parameter-part of a macrocall etc..

If a call of a not defined macro appears, the error message -def will be typed, and in case a call has fewer parameters than indicated by the formal parameter numbers of the corresponding macrodefinition, the message -par is typed, but the processing is continued, that is an undefined macro or absent parameter is treated as the empty string of symbols.

Finally the message drum (or stack) means that the capacity of the drum (or core store) has been exceeded. About 500 macronames may be used on the same blocklevel without exceeding the core store capacity.

Syntactical matrix.

	[244	(259	[* 267	(* 283	= 286
[385	ERROR	ERROR			ERROR
] 386	ERROR	ERROR		ERROR	ERROR
(387	ERROR	ERROR			ERROR
) 388	ERROR		ERROR		ERROR
* 389			ERROR		ERROR
= 390	ERROR	ERROR			
¢ 393	ERROR	ERROR	ERROR	ERROR	ERROR
OTHER SYMBOLS					SEE NOTE

NOTE: Between two = only 1 2 3 4 5 6 7 8 9 0 space and <dummy> are allowed.